Exact Algorithm for Independent Set

2017-09-12, rev. 1552dde

Independent Set

Consider an undirected unweighted graph $G = (V, E)$ and set $n = |V|$. The Maximum Independent Set problem (MIS) is to find a subset S of the vertices of largest size such that no pair of different vertices in S are connected by an edge in G. The size of the largest independent set in G is denoted by $\alpha(G)$.

MIS is NP-hard, but we will nevertheless implement algorithms that compute $\alpha(G)$ for small instance graphs G and argue both empirically and theoretically about their running time.

Notation. For a subset $X \subseteq V$, we will by $G[X]$ denote the graph *induced* by X, the graph that remains if we keep only the vertices in X and the edges among them. For a vertex v, we denote by $N[v]$ its *closed neighborhood*, i.e. v and all its neighbors. We will slightly abuse set difference and union notation by using $-$ and $+$ instead. $X + Y - Z$ for three sets X, Y, Z are those elements that are in either X or Y but not in Z.

Algorithm R_0

Consider the following simple recursive algorithm working on an input graph $G = (V, E)$. (Call it Algorithm R_0):

1. If the input graph is empty, return 0.

2. If the input graph G has a vertex v without any neighbors, return $1 + R_0(G[V - v])$. See fig. 1.

3. Otherwise find a vertex u of maximum degree (if there are several any will do) and return

$$\max(1 + R_0(G[V - N[u]]), R_0(G[V - u])),$$

keeping in mind that that $N[u]$ includes u. See fig. 2.

To see why algorithm R_0 correctly computes $\alpha(G)$, note that if there is an isolated vertex v, a maximum size independent set will always contain it (Row 2). On the other hand, if there are no isolated vertices, we can pick a vertex u of maximum degree and branch on the two cases that u is in the maximum independent set (in which case we discard all vertices adjacent to u from G), or that u is not in the maximum independent set (Row 3).
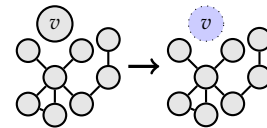

Figure 1: A graph with an isolated node v. We can safely say v is in the MIS, and continue without it.
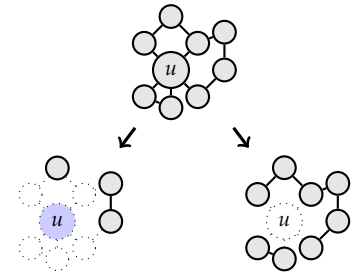

Figure 2: Vertex u has highest degree. Left branch: Assume u is in the MIS – remove u and its neighbors, since the neighbors can no longer be in the MIS. Right branch: Assume u not in the MIS – just remove u.

Inputs

The data directory contains eleven random input instances of in-
creasing size. All input files are text files on the following format
describing the undirected graph $G = (V, E)$: First comes a positive
integer n giving the number of vertices in the graph. Next follow n
rows each containing n 0/1-entries describing the so-called *adjacency*
matrix of the graph. The jth integer at the ith row is a 1 if and only if
$ij \in E$.

file	$|V|$	$\alpha(G)$
g30.in	30	14
g40.in	40	16
g50.in	50	19
g60.in	60	20
g70.in	70	22
g80.in	80	24
g90.in	89	26
g100.in	100	27
g110.in	110	29
g120.in	120	30
g130.in	130	?

Algorithm R_1

Add the following line to algorithm R_0 after its first line checking
for emptiness, and call the resulting algorithm R_1 (replacing the
recursive calls to R_0 by R_1):

If G has a vertex v of degree 1 return $1 + R_1(G[V - N[v]])$.

Before you write any code – draw (by hand) pictures such as those
above to illustrate what happens with your graph when you run
algorithm R_1 on it. Use the pictures when you implement the al-
gorithm, and when you argue the correctness of it, i.e., when you
motivate why it always computes $\alpha(G)$. You are supposed to bring
your pictures and show them to your lab assistant.

Algorithm R_2

Add the following line to algorithm R_1 after its first line checking
for emptiness, and call the resulting algorithm R_2 (replacing the
recursive calls to R_1 by R_2):

If G has a vertex v with exactly two neighbors u, w, then if $uw \in E$
return $1 + R_2(G[V - N[v]])$, else add a new vertex z to the graph
with edges to every neighbor of u and w, except v, and return $1 +
R_2(G[V - N[v] + z])$.

If G doesn't have a vertex v with exactly two neighbors, then we
proceed just as in algorithm R_1.

Draw pictures such as those above to illustrate algorithm R_2, and
use the pictures both when you implement it, and when you argue
the correctness of it, i.e., motivate why it always computes $\alpha(G)$.

Deliverables

1. Implement algorithm R_0 and run it on the instances provided in
    data/g30.txt, data/g40.txt, ..., data/g60.txt.

    Count the number of recursive calls of R_0 for each instance and
    plot the logarithm of that number vs the instance vertex size.

    What is the time complexity dependence on n?

Use whatever programming language and libraries you want, but make sure that your code is not unnecessarily long.

First implement the algorithm by making a new copy of the graph at each recursive call, just to make it as easy as possible to verify that the algorithm works as intended. Once you get this to work, you can pay attention to the problem of efficiently representing a graph through the recursion so that it is easy to remove (and add, see below) vertices to it by temporarily modifying the graph at hand.

2. After you have implemented algorithm R_1, run it on the instances data/g30.in, data/g40.in, ..., data/g100.in.

Count the number of recursive calls of R_1 for each instance and plot the logarithm of that number vs the instance vertex size.

What is the time complexity dependence on n?

3. Run algorithm R_2 on the instances data/g30.in, data/g40.in, ..., data/g120.in.

Count the number of recursive calls of R_2 for each instance and plot the logarithm of that number vs the instance vertex size.

What is the time complexity dependence on n?

4. Fill out the report on the next page; you can just use the LaTeX code if you want.

Optional

Solve independent set using bitmasks. This is a good exercise to prepare for the last lab Tree Width.

Algoritm MM

We will use a trick called *meet in the middle*, which can be used on several NP-complete problems.

Meet in the middle is similar to divide and conquer. We split the problem into two halves, solve them seperately and then merge the results. However, we can't continue to divide recursively, because the subproblems that are solved in each half are not symmetric, and also differ slightly to the original problem.

For independent set, we divide the input graph G in two halves A and B. We define a function $f : B' \rightarrow int$, where B' is a subset of B. $f(B')$ is the size of the largest subset of B' that is independent.

The subsets A' of A that are independent are saved together with their size.

Then we iterate through all the saved subsets A' and compute the maximal independent set of G where exactly the nodes A' from A are used, by querying B in the following way. If we want to use exactly the nodes A' from A we can't use any of the neighbors of A' in B. We can therefor ask f of the maximal independent set of the nodes $B' = B - N[A']$, see figure . The maximal size of an independent set $A' \cup B'$ is then $f(B') + |A'|$. The maximal independent set of G is $max_{A'}(f(B') + A')$ over all A', since this search is exhaustive over A.

This can be efficiently implemented using bitstrings and memoization over all the $2^{|B|}$ inputs to f. f can either be implemented recursively, or by just filling in a table.

Note that f can be defined recursively in the following way:
$f(B') = |B'|$ if B' is independent, else $max_{b \in B'} f(B' - b)$.

Given that $|B| < 32$, we can encode a subset of B as a 32-bit integer. We label all the nodes in B as $0, 1, 2, \ldots, |B| - 1$. Then to compute the bitstring (integer) of a subset, we sum the powers of two of the labels.

For example, if we want to find the bitstring of the subset containing nodes with labels $\{0, 3, 4, 6\}$ we compute the bitstring as $2^0 + 2^3 + 2^4 + 2^6 = 89$. In most programming languages we can use $1 \ll x$ to compute 2^x.

Discussion topics for algorithm MM

- What is the running time?

- Can you find a graph where MM is faster than R1?
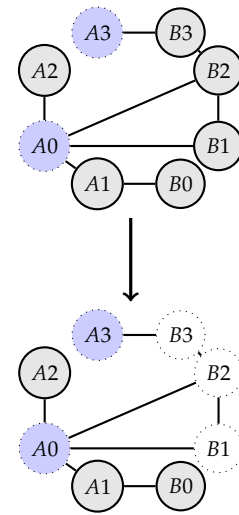
- Implementation complexity compared to R2.

Figure 3: A graph where A' is chosen as nodes $\{A0, A3\}$. Now $B' = B - N[A'] = \{B0\}$, since $\{B1, B2, B3\}$ all are connected to A'.

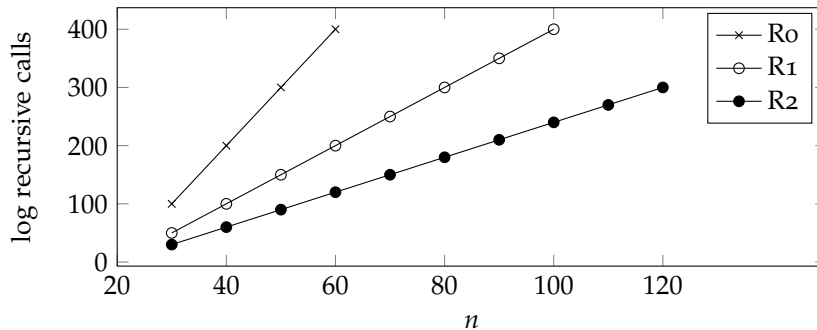## Independent Set Lab Report

by Alice Cooper and Bob Marley[1]

Correctness

Algorithm R1 correctly computes $\alpha(G)$ because [...].[2]
Algorithm R2 correctly computes $\alpha(G)$ because [...].

Empirical Running time

Experiments.   3

The running times of algorithm R_0, R_1, and R_2 appear to be
[...], [...], and [...], respectively.[4].

Theoretical Upper Bound

Denote be $T_i(n)$ the worst runtime of algorithm Ri on *any* graph on n
vertices. Note that $T_i(n)$ is a non-decreasing function of n. For R_0 we
can conclude that

$$T_0(n) \leq \max(T_0(n-1), T_0(n-1) + T_0(n-1-d_{\max}))$$
$$\leq T_0(n-1) + T_0(n-2)$$

with $d_{\max}$ the degree of the vertex we branch on. The hard part is
the one when there are no isolated vertices, in which case the vertex
u we are branching on has at least one neighbor.

For R1 we have that[5]

$$T_1(n) = [...]$$

For R2 we have that

$$T_2(n) = [...]$$

Worst Case Upper Bound   The running times of algorithm R_0, R_1, and
R_2 are in [...], [...], and [...], respectively.[6].

Perspective

This lab establishes medium skills in recursive algorithms implementation, and simple means to analyze their running time.

The algorithms investigated here can hardly be called "advanced", but the main message is that no one knows how to do significantly better with any other means. Indeed, the strongest theoretical worst case run time bound for maximum independent set to date is obtained by a computerized calculation on a huge set of branching reduction rules, just like the ones we've looked at here [Robson. Algorithms for maximum independent sets. Journal of Algorithms, 7(3):425–440, 1986]. For a much cleaner analysis with just a few rules, obtaining a comparative bound, see [Fomin, Grandoni, and Kratsch. A measure & conquer approach for the analysis of exact algorithms, JACM 56 (5), article No. 25, 2009].